Nama : RIZQILLAH
NIM : 1957301020
MK : Pemrograman Lanjut

ALGORITMA SORTING

1.1 Tujuan

Sorting adalah proses menyusun elemen – elemen dengan tata urut tertentu dan proses tersebut terimplementasi dalam bermacam aplikasi. Kita ambil contoh pada aplikasi perbankan. Aplikasi tersebut mampu menampilkan daftar account yang aktif. Hampir seluruh pengguna pada sistem akan memilih tampilan daftar berurutan secara ascending demi kenyamanan dalam penelusuran data.

Beberapa macam algoritma sorting telah dibuat karena proses tersebut sangat mendasar dan sering digunakan. Oleh karena itu, pemahaman atas algoritma – algoritma yang ada sangatlah berguna.

Setelah menyelesaikan pembahasan pada bagian ini, anda diharapkan mampu :

1. Memahami dan menjelaskan algoritma dari insertion sort, selection sort, merge sort dan quick sort.
2. Membuat implementasi pribadi menggunakan algoritma yang ada

1.2 Insertion Sort

Salah satu algoritma *sorting* yang paling sederhana adalah *insertion sort*. Ide dari algoritma ini dapat dianalogikan seperti mengurutkan kartu. Penjelasan berikut ini menerangkan bagaimana algoritma *insertion sort* bekerja dalam pengurutan kartu. Anggaplah anda ingin mengurutkan satu set kartu dari kartu yang bernilai paling kecil hingga yang paling besar. Seluruh kartu diletakkan pada meja, sebutlah meja ini sebagai meja pertama, disusun dari kiri ke kanan dan atas ke bawah. Kemudian kita mempunyai meja yang lain, meja kedua, dimana kartu yang diurutkan akan diletakkan. Ambil kartu pertama yang terletak pada pojok kiri atas meja pertama dan letakkan pada meja kedua. Ambil kartu kedua dari meja pertama, bandingkan dengan kartu yang berada pada meja kedua, kemudian letakkan pada urutan yang sesuai setelah perbandingan. Proses tersebut akan berlangsung hingga seluruh kartu pada meja pertama telah diletakkan berurutan pada meja kedua.

Insertion Sort merupakan metode pengurutan dimana langkah-langkahnya ialah membandingkan dari data kedua ke data pertama (Data-N <==> Data-N-1). Berbeda dengan Bubble Sort dimana, proses perbandingannya dimulai dari Data Pertama sampai Data Terakhir.

Algoritma *insertion sort* pada dasarnya memilah data yang akan diurutkan menjadi dua bagian, yang belum diurutkan (meja pertama) dan yang sudah diurutkan (meja kedua). Elemen pertama diambil dari bagian array yang belum diurutkan dan kemudian diletakkan sesuai posisinya pada bagian lain dari array yang telah diurutkan. Langkah ini dilakukan secara berulang hingga tidak ada lagi elemen yang tersisa pada bagian array yang belum diurutkan.

Keuntungan menggunakan *Insertion Sort* adalah :

- Dapat diimplementasikan dengan simpel.
- Sangat efisien untuk data berukuran kecil.
- *Insertion sort* dapat langsung menyortir list data ketika menerima input.
- Secara praktikal lebih efisien dibandingkan dengan *selection* dan *bubble sort*.

1.2.1 Algoritma

```
void insertionSort(Object array[], int startIdx, int endIdx) {
    for (int i = startIdx; i < endIdx; i++) {
         int k = i;
         for (int j = i + 1; j < endIdx; j++) {
           if (((Comparable) array[k]).compareTo(array[j]) > 0) {
               k = j;
           }
         }
         swap(array[i], array[k]);
    }
}
```

1.2.2 Sebuah Contoh

Data	1st Pass	2nd Pass	3rd Pass	4th Pass
Mango	Mango	Apple	Apple	Apple
Apple	Apple	Mango	Mango	Banana
Peach	Peach	Peach	Orange	Mango
Orange	Orange	Orange	Peach	Orange
Banana	Banana	Banana	Banana	Peach

Gambar 1. Contoh insertion sort

Pada akhir modul ini, anda akan diminta untuk membuat implementasi bermacam algoritma sorting yang akan dibahas pada bagian ini.

Berikut ini gambaran dari implementasi *Insertion Sort* :

1st Cycle:

(70, 60, 30, 50, 40, 20) -> (60, 70, 30, 50, 40, 20)

(**60**, **70**, 30, 50, 40, 20)

Pertama melakukan perbandingan antara nilai array pada index 1 dengan array index 0. Dan melakukan swap jika hasil perbandingan yang diinginkan tercapai.

2nd Cycle:

(**60**, **70**, 30, 50, 40, 20) -> (**60**, 30, **70**, 50, 40, 20)

(**60**, 30, **70**, 50, 40, 20) -> (30, **60**, **70**, 50, 40, 20)

(**30**, **60**, **70**, 50, 40, 20)

- Mengambil nilai array index 2 dan melakukan perbandingan dengan index 1. Jika perbandingan tercapai, maka melakukan swap.
- Mengambil nilai index 1 dan index 0, kemudian melakukan perbandingan. Jika kondisi tercapai, maka melakukan swap.

3rd Cycle:

(**30**, **60**, **70**, 50, 40, 20) -> (**30**, **60**, 50, **70,** 40, 20)

(**30**, **60**, 50, **70,** 40, 20) -> (**30**, 50, **60**, **70**, 40, 20)

(**30**, 50, **60**, **70**, 40, 20) -> (**30**, 50, **60**, **70**, 40, 20)

(**30**, **50**, **60**, **70**, 40, 20)

- Mengambil nilai index 3, dan melakukan perbandingan dengan index 2. Jika perbandingan tercapai, maka melakukan swap.
- Mengambil nilai index 2 dan index 1. Kemudian melakukan perbandingan, jika kondisi tercapai maka melakukan swap.
- Mengambil nilai index 1 dan index 0. Dikarenakan nilai index 1 lebih besar ketimbang index 0, maka tidak akan melakukan swap. Dan program lanjut ke selanjutnya.

4th Cycle:

(**30**, **50**, **60, 70,** 40, 20) -> (**30**, **50**, **60,** 40, **70**, 20)

(**30**, **50**, **60**, 40, **70**, 20) -> (**30**, **50**, 40, **60**, **70**, 20)

(**30**, **50**, 40, **60**, **70**, 20) -> (**30**, 40, **50**, **60**, **70**, 20)

(**30**, 40, **50**, **60**, **70**, 20) -> (**30**, 40, **50**, **60**, **70**, 20)

(**30**, **40**, **50**, **60**, **70**, 20)

- Mengambil nilai index 4, dan melakukan perbandingan dengan index 3. Jika perbandingan tercapai, maka melakukan swap.
- Mengambil nilai index 3 dan index 2. Dan melakukan perbandingan, jika kondisi tercapai, maka melakukan swap.
- Mengambil nilai index 2 dan index 1. Dan melakukan perbandingan, jika kondisi tercapai, maka melakukan swap.
- Dan mengambil nilai index 1 dan 0. Dan melakukan perbandingan, dikarenakan nilai index 1 lebih besar dari index 0, maka tidak akan melakukan swap.

5th Cycle:

(**30**, **40**, **50**, **60**, **70**, 20) -> (**30**, **40**, **50**, **60**, 20, **70**)

(**30**, **40**, **50**, **60**, 20, **70**) -> (**30**, **40**, **50**, 20, **60**, **70**)

(**30**, **40**, **50**, 20, **60**, **70**) -> (**30**, **40**,20, **50**, **60**, **70**)

(**30**, **40**, 20, **50**, **60**, **70**) -> (**30**, 20, **40**, **50**, **60**, **70**)

(**30**, 20, **40**, **50**, **60**, **70**) -> (20, **30**, **40**, **50**, **60**, **70**)

(**20**, **30**, **40**, **50**, **60**, **70**)

- Mengambil nilai index 5, dan melakukan perbandingan dengan index 4. Jika perbandingan tercapai, maka melakukan swap.
- Mengambil nilai index 4 dan index 3. Dan melakukan perbandingan, jika kondisi tercapai, maka melakukan swap.
- Mengambil nilai index 3 dan index 2. Dan melakukan perbandingan, jika kondisi tercapai, maka melakukan swap.
- Mengambil nilai index 2 dan index 1. Dan melakukan perbandingan, jika kondisi tercapai, maka melakukan swap.
- Mengambil nilai index 1 dan index 0. Dan melakukan perbandingan, jika kondisi tercapai, maka melakukan swap. Dan didapatkanlah hasil akhir dengan nilai yang sudah ter-sortir.

1.2.3 Contoh Program

```
package InsertionSort;

public class InsertionSort {
    public static void insertionSort(int array[]) {
        int n = array.length;
        for (int j = 1; j < n; j++) {
            int key = array[j];
            int i = j - 1;
            while ((i > -1) && (array[i] > key)) {
                array[i + 1] = array[i];
```

```
                i--;
            }
            array[i + 1] = key;
        }
    }

    public static void main(String[] args) {
        int[] arr1 = {90, 40, 50, 20, 10, 80, 70, 60, 30};
        System.out.println("Before Insertion Sort");
        for (int i : arr1) {
            System.out.print(i + " ");
        }
        System.out.println();

        insertionSort(arr1);

        System.out.println("After Insertion Sort");
        for (int i : arr1) {
            System.out.print(i + " ");
        }
    }
}
```

Hasil :

```
Before Insertion Sort
90 40 50 20 10 80 70 60 30
After Insertion Sort
10 20 30 40 50 60 70 80 90 BUILD SUCCESSFUL (total time: 0 seconds)
```

1.3 Selection Sort

Selection sort adalah algoritma untuk sorting sama seperti halnya dengan *Insertion sort* yang simpel, namun tidak efisien. Pada perulangan pertama, metode ini akan memilih elemen terkecil dari array, kemudian menukarnya dengan elemen pertama. Selanjutnya pada perulangan kedua, akan memilih nilai terkecil kedua (atau elemen dengan nilai terkecil diantara seluruh elemen yang tersisa), dan kemudian menukarnya dengan elemen yang kedua.

Dibanding *Bubble Sort*, *Selection Sort* jelas lebih baik dari segi kecepatan proses pengurutannya. Karena, Inti dari algoritma *Selection Sort* ialah mencari nilai yang paling kecil(Jika *Ascending*) atau nilai yang paling besar(Jika *Descending*) di urutan Data.

Selection sort merupakan perbaikan dari metode *bubble sort* dengan mengurangi jumlah perbandingan. *Selection sort* merupakan metode pengurutan dengan mencari nilai data terkecil dimulai dari data diposisi 0 hingga diposisi N-1.

Algoritma *selection sort* akan memindai nilai terkecil dari suatu kumpulan data dan jika ada, data tersebut akan diletakkan pada urutan pertama. Begitu selanjutnya untuk urutan kedua dan seterusnya.

Dikatakan *selection sort* karena algoritma ini mencoba memilih satu per satu elemen data dari posisi awal, untuk mencari data paling kecil dengan mencatat posisi index-nya saja, lalu dilakukan pertukaran hanya sekali pada akhir setiap tahapan. Algoritma *Selection Sort* dilakukan untuk menyempurnakan kekurangan dari *bubble sort* yang melakukan pertukaran setiap kali perbandingan memenuhi kriterianya.

Jika anda diminta untuk membuat algoritma sorting tersendiri, anda mungkin akan menemukan sebuah algoritma yang mirip dengan *selection sort*. Layaknya *insertion sort*, algoritma ini sangat rapat dan mudah untuk diimplementasikan.

Mari kita kembali menelusuri bagaimana algoritma ini berfungsi terhadap satu paket kartu. Asumsikan bahwa kartu tersebut akan diurutkan secara *ascending*. Pada awalnya, kartu tersebut akan disusun secara linier pada sebuah meja dari kiri ke kanan, dan dari atas ke bawah. Pilih nilai kartu yang paling rendah, kemudian tukarkan posisi kartu ini dengan kartu yang terletak pada pojok kiri atas meja. Lalu cari kartu dengan nilai paling rendah diantara sisa kartu yang tersedia. Tukarkan kartu yang baru saja terpilih dengan kartu pada posisi kedua. Ulangi langkah–langkah tersebut hingga posisi kedua sebelum posisi terakhir dibandingkan dan dapat digeser dengan kartu yang bernilai lebih rendah.

Ide utama dari algoritma *selection sort* adalah memilih elemen dengan nilai paling rendah dan menukar elemen yang terpilih dengan elemen ke-i. Nilai dari i dimulai dari 1 ke n, dimana n adalah jumlah total elemen dikurangi 1.

1.3.1 Algoritma

```
void selectionSort(Object array[], int startIdx, int endIdx) {
   int min;
   for (int i = startIdx; i < endIdx; i++) {
       min = i;
       for (int j = i + 1; j < endIdx; j++) {
         if (((Comparable) array[min]).compareTo(array[j]) > 0) {
             min = j;
         }
       }
       swap(array[min], array[i]);
    }
}
```

1.3.2 Sebuah Contoh

Data	1st Pass	2nd Pass	3rd Pass	4th Pass
Maricar	Hannah	Hannah	Hannah	Hannah
Vanessa	*Vanessa*	Margaux	Margaux	Margaux
Margaux	*Margaux*	*Vanessa*	Maricar	Maricar
Hannah	Maricar	*Maricar*	*Vanessa*	Rowena
Rowena	Rowena	Rowena	*Rowena*	Vanessa

Gambar 2. Contoh selection sort

Berikut gambaran dari implementasi *Selection Sort* :

1st Cycle:

(70, 60, 30, 50, 40,20) ->min 60

(70, 60, 30, 50, 40, 20)->min 30

(70, 60, 30, 50, 40, 20)->min 30

(70, 60, 30, 50, 40, 20)->min 30

(70, 60, 30, 50, 40, 20)->min 20

(**20**, 60, 30, 50, 40, 70) -> swap (70,20)

- Proses diatas adalah menyeleksi nilai terkecil. Nilai pertama didapat adalah 60, kemudian mencari nilai selanjutnya dan membandingkan dengan sebelumnya, dan jika terkecil maka akan mengambil nilai tersebut sebagai terkecil sampai akhirnya menemukan nilai yang benar-benar terkecil dari list dan memindahkannya ke paling depan secara ascending.

2nd Cycle:

(**20**, 60, 30, 50, 40, 70) ->min 30

(**20**, 60, 30, 50, 40, 70) ->min 30

(**20**, 60, 30, 50, 40, 70) ->min 30

(**20**, 60, 30, 50, 40, 70) ->min 30

(**20**, **30**, 60, 50, 40, 70) ->swap (60,30)

- Mengambil nilai 30 sebagai terkecil, dan mencari nilai terkecil lainnya, dikarenakan tidak ada, maka nilai 30 akan ditempatkan pada index setelah proses pertama.

3rd Cycle:

(**20**, **30**, 60, 50, 40, 70) -> min 50

(**20**, **30**, 60, 50, 40, 70) -> min 40

(**20**, **30**, 60, 50, 40, 70) -> min 40

(**20**, **30**, **40**, 50, 60, 70) -> swap(60,40)

- Mengambil nilai 50 sebagai terkecil, dan mencari nilai selanjutnya dan mendapat nilai 40. Dan mencari nilai selanjutnya, dikarenakan 40 terkecil, maka akan menempatkan pada index setelah proses sebelumnya.

4th Cycle:

(**20**, **30**, **40**, 50, 60, 70) -> min 50

(**20**, **30**, **40**, 50, 60, 70) -> min 50

(**20**, **30**, **40**, **50**, 60, 70)

- Mengambil nilai 50 sebagai terkecil. Dan mencari nilai lainnya, dikarenakan 50 nilai terkecil yang tinggal, maka akan dipindahkan ke index setelah operasi sebelumnya.

5th Cycle:

(**20**, **30**, **40**, **50**, 60, 70) -> min 60

(**20**, **30**, **40**, **50**, **60**, **70**)

- Mengambil nilai 60 sebagai terkecil. Dan dikarenakan nilai tinggal 2, oleh karenanya hanya akan terjadi perbandingan, dan nilai terkecil diletakkan disebelah nilai sebelumnya.

1.3.3 Contoh Program

```
package SelectionSort;
public class SelectionSort {
    public static void selectionSort(int[] arr) {
        for (int i = 0; i < arr.length - 1; i++) {
            int index = i;
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[j] < arr[index]) {
                    index = j;
                }
            }
            int smallerNumber = arr[index];
            arr[index] = arr[i];
            arr[i] = smallerNumber;
        }
    }
```

```
    public static void main(String args[]) {
        int[] arr1 = {70, 60, 30, 50, 40, 20};
        System.out.println("Before Selection Sort");
        for (int i : arr1) {
            System.out.print(i + " ");
        }
        System.out.println();

        selectionSort(arr1);

        System.out.println("After Selection Sort");
        for (int i : arr1) {
            System.out.print(i + " ");
        }
    }
}
```

Hasil :

```
Before Selection Sort
70 60 30 50 40 20
After Selection Sort
20 30 40 50 60 70 BUILD SUCCESSFUL (total time: 0 seconds)
```

1.4 Merge Sort

Merge sort merupakan algoritma pengurutan dalam ilmu komputer yang dirancang untuk memenuhi kebutuhan pengurutan atas suatu rangkaian data yang tidak memungkinkan untuk ditampung dalam memori komputer karena jumlahnya yang terlalu besar. Algoritma ini ditemukan oleh John von Neumann pada tahun 1945.

Algoritma pengurutan data *merge sort* dilakukan dengan menggunakan cara *divide and conquer* yaitu dengan memecah kemudian menyelesaikan setiap bagian kemudian menggabungkannya kembali. Pertama data dipecah menjadi 2 bagian dimana bagian pertama merupakan setengah (jika data genap) atau setengah minus satu (jika data ganjil) dari seluruh data, kemudian dilakukan pemecahan kembali untuk masing-masing blok sampai hanya terdiri dari satu data tiap blok.

Setelah itu digabungkan kembali dengan membandingkan pada blok yang sama apakah data pertama lebih besar daripada data ke-tengah+1, jika ya maka data ke-tengah+1 dipindah sebagai data pertama, kemudian data ke-pertama sampai ke-tengah digeser menjadi data ke-dua sampai ke-tengah+1, demikian seterusnya sampai menjadi satu blok utuh seperti awalnya. Sehingga metode *merge sort* merupakan metode yang membutuhkan fungsi rekursi untuk penyelesaiannya.

1.4.1 Memahami Merge Sort

Algoritma dirumuskan dalam 3 langkah berpola *divide-and-conquer*. Berikut menjelaskan langkah kerja dari *Merge sort*.

1. *Divide*, Memilah elemen – elemen dari rangkaian data menjadi dua bagian.
2. *Conquer*, Conquer setiap bagian dengan memanggil prosedur *merge sort* secara rekursif
3. Kombinasi, Mengkombinasikan dua bagian tersebut secara rekursif untuk mendapatkan rangkaian data berurutan

Proses rekursi berhenti jika mencapai elemen dasar. Hal ini terjadi bilamana bagian yang akan diurutkan menyisakan tepat satu elemen. Sisa pengurutan satu elemen tersebut menandakan bahwa bagian tersebut telah terurut sesuai rangkaian.

1.4.2 Pola Divide and Conquer

Algoritma *Divide and Conquer* merupakan algoritma yang sangat populer di dunia Ilmu Komputer. *Divide and Conquer* merupakan algoritma yang berprinsip memecah-mecah permasalahan yang terlalu besar menjadi beberapa bagian kecil sehingga lebih mudah untuk diselesaikan.

Algoritma *Divide and Conquer* adalah strategi pemecahan masalah yang besar dengan cara melakukan pembagian masalah yang besar tersebut menjadi beberapa bagian yang lebih kecil secara rekursif hingga masalah tersebut dapat dipecahkan secara langsung. Solusi yang didapat dari setiap bagian kemudian digabungkan untuk membentuk sebuah solusi yang utuh.

Beberapa algoritma mengimplementasikan konsep rekursi untuk menyelesaikan permasalahan. Permasalahan utama kemudian dipecah menjadi sub-masalah, kemudian solusi dari sub-masalah akan membimbing menuju solusi permasalahan utama.

Pada setiap tingkatan rekursi, pola tersebut terdiri atas 3 langkah.

1. Divide
   Memilah masalah menjadi sub masalah
2. Conquer
   Selesaikan sub masalah tersebut secara rekursif. Jika sub-masalah tersebut cukup ringkas dan sederhana, pendekatan penyelesaian secara langsung akan lebih efektif
3. Kombinasi
   Mengkombinasikan solusi dari sub-masalah, yang akan membimbing menuju penyelesaian atas permasalahan utama

Algoritma *divide and conquer* mempunyai kelebihan yang dapat mengurangi kompleksitas pencarian solusi suatu masalah karena prinsip kerjanya yang membagi-bagi masalah menjadi upmasalah-upmasalah yang lebih kecil. Kelebihan tersebut banyak menguntungkan dari segi waktu, tenaga, dan sumberdaya. Salah satu penerapan dari algoritma ini adalah pada mekanisme komputer (atau mesin) paralel. Algoritma *Divide And Conquer* terbukti menampilkan hasil yang paling baik dan paling sesuai untuk komputer dengan hirarki memori tinggi serta memiliki *cache*.

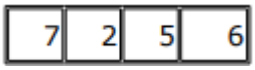
Cache merupakan memori sementara komputer yang berfungsi untuk mempersingkat pengaksesan data. Masalah yang dihadapi komputer paralel antara lain adalah bagaimana membuat *cache* bekerja lebih efisien pada tiap-tiap komputer yang terhubung secara paralel. Tanpa strategi tertentu, kinerja *cache* pada komputer paralel akan memakan waktu yang lama karena ketika menemui suatu masalah, pemecahan harus dilakukan secara bergantian, satu per satu. Sedangkan komputer paralel menuntut semua perangkatnya mampu bekerja secara paralel. Paralelitas tersebut dapat diatasi dengan penerapan algoritma *Divide And Conquer* yang memungkinkan pemecahan masalah secara independen, namun tetap konkuren.

1.4.3 Algoritma

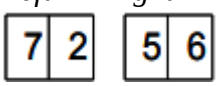
```
void mergeSort(Object array[], int startIdx, int endIdx) {
    if (array.length != 1) {
        //Membagi rangkaian data, rightArr dan leftArr
        mergeSort(leftArr, startIdx, midIdx);
        mergeSort(rightArr, midIdx + 1, endIdx);
        combine(leftArr, rightArr);
    }
}
```

1.4.4 Sebuah Contoh

Rangkaian data:

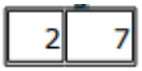


Membagi rangkaian menjadi dua bagian:
LeftArr RightArr



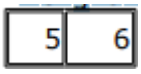
Membagi *LeftArr* menjadi dua bagian:
LeftArr RightArr

Mengkombinasikan

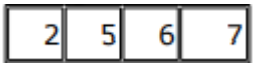


Membagi *RightArr* menjadi dua bagian:
LeftArr RightArr

Mengkombinasikan



Mengkombinasikan *LeftArr* dan *RightArr*.



Gambar 3. Contoh merge sort

1.4.5 Contoh Program

```
package MergeSort;
public class MergeSort {
    void merge(int arr[], int beg, int mid, int end) {
        int l = mid - beg + 1;
        int r = end - mid;
        int LeftArray[] = new int[l];
        int RightArray[] = new int[r];

        for (int i = 0; i < l; ++i) {
            LeftArray[i] = arr[beg + i];
        }

        for (int j = 0; j < r; ++j) {
            RightArray[j] = arr[mid + 1 + j];
        }
        int i = 0, j = 0;
        int k = beg;
        while (i < l && j < r) {
            if (LeftArray[i] <= RightArray[j]) {
                arr[k] = LeftArray[i];
                i++;
            } else {
                arr[k] = RightArray[j];
                j++;
```

```
            }
            k++;
        }
        while (i < l) {
            arr[k] = LeftArray[i];
            i++;
            k++;
        }
        while (j < r) {
            arr[k] = RightArray[j];
            j++;
            k++;
        }
    }

    void sort(int arr[], int beg, int end) {
        if (beg < end) {
            int mid = (beg + end) / 2;
            sort(arr, beg, mid);
            sort(arr, mid + 1, end);
            merge(arr, beg, mid, end);
        }
    }

    public static void main(String args[]) {
        int arr[] = {7, 2, 5, 6};
        MergeSort ob = new MergeSort();
        System.out.println("Array Before Sorted");
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }

        ob.sort(arr, 0, arr.length - 1);
        System.out.println("\nArray After Sorted");
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}
```

Hasil :

```
Array Before Sorted
7 2 5 6
Array After Sorted
2 5 6 7 BUILD SUCCESSFUL (total time: 0 seconds)
```

1.5 Quicksort

Quick Sort adalah algoritma yang dijalankan sebagai akibat dari terlalu banyaknya daftar yang diurutkan dengan keefektifan O(n log n), Algoritma dibuat tahun 1960 oleh Tony Hoare dan dianalisa lebih lanjut oleh Robert Sedgewick dalam thesisnya. *Quicksort* mendapatkan adaptasi yang luas contohnya Unix menggunakannya sebagai fungsi pengurutan library secara *default*. *Quick sort* secara teori adalah mempartisi data dengan menemukan pivot (data ditengah) kemudian menggabungkannya (hampir sama seperti *merge sort*).

Proses pengurutan dilakukan dengan memecah kumpulan data menjadi dua bagian berdasarkan nilai pivot yang dipilih. Pada prinsipnya nilai pivot yang dipilih ini akan ditempatkan pada posisinya disetiap akhir proses partisi. Setelah proses partisi selesai dan menempatkan pivot pada posisinya yang tepat maka proses pengurutan dilanjutkan secara rekursif untuk mengurutkan data bagian kiri dari pivot dan bagian kanan dari pivot tersebut.

Algoritma *quicksort* memiliki *worst-case running time* - n^2 pada array input n angka. Terlepas dari *worst-case running time*, *quicksort* sering merupakan pilihan praktis terbaik untuk menyortir karena sangat efisien secara rata-rata: *Running time* yang diharapkan adalah (n log n), dan faktor konstan yang tersembunyi di (n log n) notasi cukup kecil. Ini juga memiliki keuntungan penyortiran di tempat dan bekerja dengan baik bahkan di lingkungan virtual-memory.

Algoritma ini dengan subrutin sangat penting digunakan dengan *quicksort* untuk partisi. Versi *quicksort* juga menggunakan sampling acak. Algoritma ini memiliki waktu berjalan yang diharapkan baik, dan tidak ada masukan tertentu yang memunculkan perilaku terburuknya. *Quick sort* memiliki algoritma acak sehingga menunjukkan bahwa ia berjalan dalam (n^2) waktu dalam kasus terburuk dan, dengan asumsi elemen yang berbeda, dalam harapan O(n log n) waktu.

Quicksort ditemukan oleh C.A.R Hoare. Seperti pada *merge sort*, algoritma ini juga berdasar pada pola *divide-and-conquer*. Berbeda dengan merge sort, algoritma ini hanya mengikuti langkah – langkah sebagai berikut :

1. Divide

   Memilah rangkaian data menjadi dua sub-rangkaian A[p…q-1] dan A[q+1…r] dimana setiap elemen A[p…q-1] adalah kurang dari atau sama dengan A[q] dan setiap elemen pada A[q+1…r] adalah lebih besar atau sama dengan elemen pada A[q]. A[q] disebut sebagai elemen pivot. Perhitungan pada elemen q merupakan salah satu bagian dari prosedur pemisahan.

2. Conquer
   Mengurutkan elemen pada sub-rangkaian secara rekursif

Pada algoritma *quicksort*, langkah "kombinasi" tidak di lakukan karena telah terjadi pengurutan elemen–elemen pada sub-array

Partitioning

Partitioning adalah sebuah proses untuk membuat sebuah untaian nilai terpisah berdasarkan suatu nilai yang ditetapkan sebagai pembatas (pivot). Hasil dari proses ini adalah seluruh elemen yang nilainya lebih kecil dari pivot ada pada sisi yang berlawanan dengan nilai yang lebih besar dari pivot.

Rekursif

Pada algoritma *quick sort*, proses *partitioning* kemudian dipanggil dua kali lagi, untuk untaian nilai yang ada di sebelah kanan dan untuk untaian nilai yang ada di sebelah kiri. Hal ini dilakukan berulang-ulang hingga hanya tersisa satu elemen untuk dipartisi, yang tentunya sudah tidak diperlukan lagi.

Cara Pemilihan Pivot

Pivot adalah elemen krusial dalam algoritma *quick sort*. jika salah memilihnya, maka algoritma yang dijalankan tidak akan secepat yang kita inginkan. jadi dengan kata lain, pivot ini menentukan performa dari algoritma ini, ada 3 cara memilih pivot, yaitu:

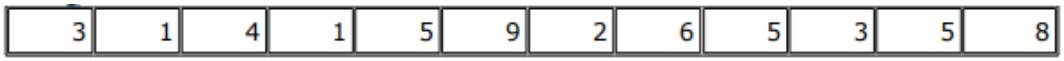
- Pivot berasal dari elemen pertama atau akhir dari sebuah list. Pivot jenis ini bisa kita gunakan apabila List yang akan disusun memiliki angka yang benar-benar acak. dan sebaliknya pivot jenis ini sangat buruk jika kita memiliki list yang sebagian/sepenuhnya sudah tersusun.
- Pivot dipilih secara acak, pivot jenis ini sering digunakan bila kita tidak mengenal List yang akan di susun, hasilnya pun akan random, terkadang performanya baik, namun terkadang performanya juga buruk.
- Pivot berasal dari median tabel. cara ini yang paling sering digunakan, karena pivot ini bersifat seimbang dimana semua elemen akan dibagi rata dan hasilnya pun cukup baik untuk data yang sebagian sudah di sort ataupun semua data masih acak.

1.5.1 Algoritma

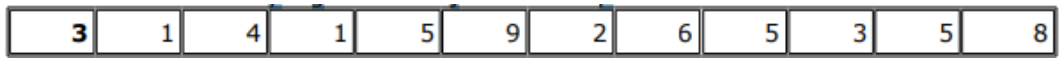
```
void quickSort(Object array[], int leftIdx, int rightIdx) {
    int pivotIdx;
    /* Kondisi Terminasi */
    if (rightIdx > leftIdx) {
        pivotIdx = partition(array, leftIdx, rightIdx);
        quickSort(array, leftIdx, pivotIdx - 1);
        quickSort(array, pivotIdx + 1, rightIdx);
    }
}
```

1.5.2 Sebuah Contoh

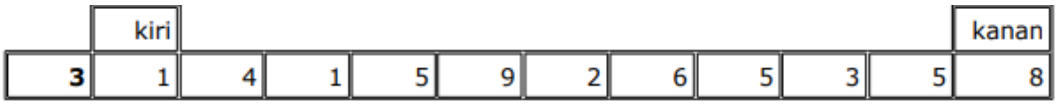
Rangkaian data:



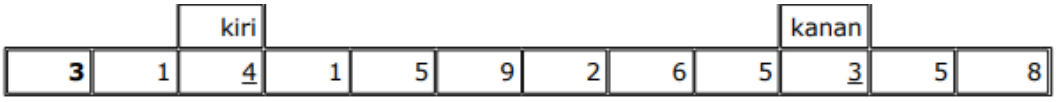
Pilih sebuah elemen yang akan menjadi elemen pivot.



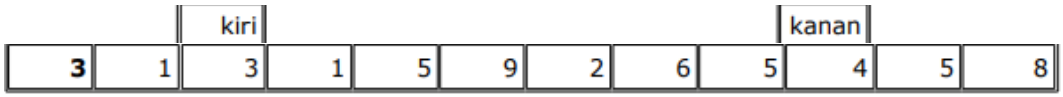
Inisialisasi elemen kiri sebagai elemen kedua dan elemen kanan sebagai elemen akhir



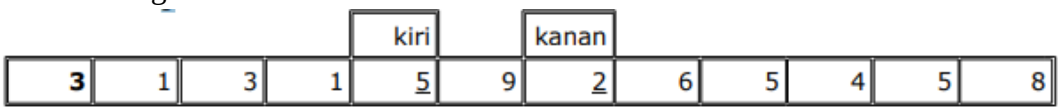
Geser elemen kiri kearah kanan sampai ditemukan nilai yang lebih besar dari elemen pivot tersebut. Geser elemen kanan ke arah kiri sampai ditemukan nilai dari elemen yang tidak lebih besar dari elemen tersebut.



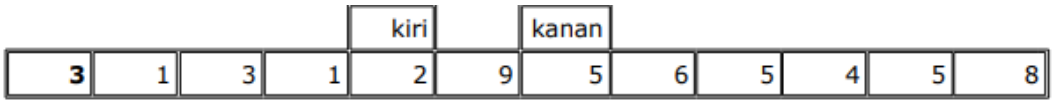
Tukarkan antara elemen kiri dan kanan



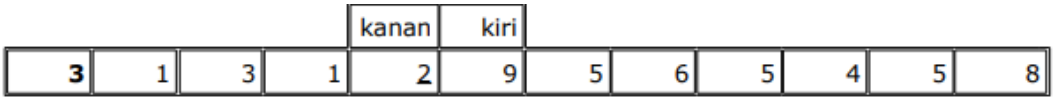
Geserkan lagi elemen kiri dan kanan.



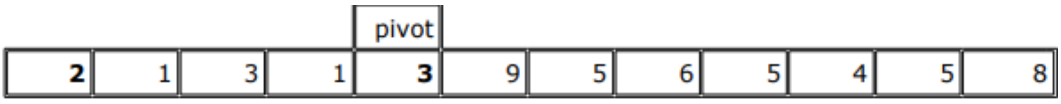
Tukarkan antar elemen kembali.



Geserkan kembali elemen kiri dan kanan.



Terlihat bahwa titik kanan dan kiri telah digeser sehingga mendapatkan nilai elemen kanan < elemen kiri. Dalam hal ini tukarkan elemen pivot dengan elemen kanan.



Gambar 4. Contoh quicksort

Kemudian urutkan elemen sub-rangkaian pada setiap sisi dari elemen pivot.

Kelebihan Algoritma Quick Sort

- Karena sorting langsung dilakukan di array asli, maka tidak memerlukan memory tambahan.
- Performanya tinggi

Kelemahan Algoritma Quick Sort

- Jika salah memilih pivot, maka algoritmanya akan sangat buruk.
- Sulit untuk dimengerti.

1.5.3 Contoh Program

```
package QuickSort;
public class QuickSort {
    public static int partition(int arr[], int beg, int end) {
        int left, right, temp, loc, flag;
        loc = left = beg;
        right = end;
        flag = 0;
        while (flag != 1) {
            while ((arr[loc] <= arr[right]) && (loc != right)) {
                right--;
            }
            if (loc == right) {
                flag = 1;
            } else if (arr[loc] > arr[right]) {
                temp = arr[loc];
```

```
                arr[loc] = arr[right];
                arr[right] = temp;
                loc = right;
            }
            if (flag != 1) {
                while ((arr[loc] >= arr[left]) && (loc != left)){
                    left++;
                }
                if (loc == left) {
                    flag = 1;
                } else if (arr[loc] < arr[left]) {
                    temp = arr[loc];
                    arr[loc] = arr[left];
                    arr[left] = temp;
                    loc = left;
                }
            }
        }
        return loc;
    }

    static void quickSort(int arr[], int beg, int end) {
        int loc;
        if (beg < end) {
            loc = partition(arr, beg, end);
            quickSort(arr, beg, loc - 1);
            quickSort(arr, loc + 1, end);
        }
    }

    public static void main(String[] args) {
        int[] arr = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8};
        System.out.println("Array Before Sorted:");
        for (int i : arr) {
            System.out.print(i + " ");
        }

        quickSort(arr, 0, arr.length - 1);
        System.out.println("\nArray After Sorted:");
        for (int i : arr) {
            System.out.print(i + " ");
        }
    }
}
```

Hasil :

```
Array Before Sorted:
3 1 4 1 5 9 2 6 5 3 5 8
Array After Sorted:
1 1 2 3 3 4 5 5 5 6 8 9 BUILD SUCCESSFUL (total time: 0 seconds)
```

1.6 Latihan

1.6.1 Insertion Sort

Impelementasikan algoritma insertion sort dalam Java untuk mengurutkan serangkaian data integer. Lakukan percobaan terhadap hasil implementasi anda terhadap rangkaian data integer yang dimasukkan oleh pengguna melalui command line.

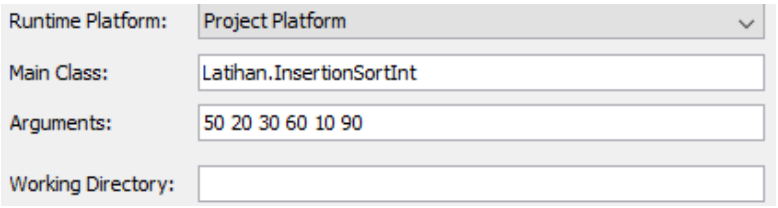
Program :

```
package Latihan;
public class InsertionSortInt {
    public static void insertionSort(int array[]) {
        int n = array.length;
        for (int j = 1; j < n; j++) {
            int key = array[j];
            int i = j - 1;
            while ((i > -1) && (array[i] > key)) {
                array[i + 1] = array[i];
                i--;
            }
            array[i + 1] = key;
        }
    }

    public static void main(String[] args) {
        int[] arr = new int[args.length];
        for (int i = 0; i < args.length; i++) {
            arr[i] = Integer.parseInt(args[i]);
        }
        System.out.println("Sebelum Insertion Sort");
        for (int i : arr) {
            System.out.print(i + " ");
        }
        System.out.println();

        insertionSort(arr);

        System.out.println("Setelah Insertion Sort");
        for (int i : arr) {
            System.out.print(i + " ");
        }
    }
}
```

Hasil :

```
Sebelum Insertion Sort
50 20 30 60 10 90
Setelah Insertion Sort
10 20 30 50 60 90 BUILD SUCCESSFUL (total time: 0 seconds)
```

Arguments :



Analisis :

Proses diatas diawali dengan user menginput data pada arguments melalui command-line. Kemudian program akan mencoba membuat array dengan panjang sebanyak data yang diinputkan. Kemudian data yang diinputkan tersebut akan dimasukkan kedalam array yang akan menyimpan data yang akan di sortir.

```
for (int i : arr) {
     System.out.print(i + " ");
}
```

- Proses diatas adalah forEach yang berguna untuk melakukan perulangan sebanyak nilai array dan akan mencetak data yang ada didalam array tersebut.

```
insertionSort(arr);
```

- kemudian program mencoba memanggil fungsi insertionSort yang diisi dengan data dari array yang ada.

```
int n = array.length;
```

- pada fungsi insertionSort, dibuat sebuah variabel yang menyimpan panjang dari data array yang diterima.

```
for (int j = 1; j < n; j++) {
```

- melakukan perulangan nilai j = 1, dan j lebih kecil dari panjang data array.

```
    int key = array[j];
    int i = j - 1;
```

- membuat variabel key yang menyimpan nilai dari array pada index j.
- Dan membuat variabel i yang menyimpan nilai dari j – 1.

```
while ((i > -1) && (array[i] > key)) {
      array[i + 1] = array[i];
      i--;
}
```

- mencoba melakukan perulangan selama nilai i lebih besar dari -1, dan nilai dari array pada index i lebih besar dari nilai key.
- Menyimpan nilai array pada index i kedalam array pada index i + 1. Dan i--.

```
array[i + 1] = key;
```

- mengisi nilai dari array pada index i + 1 dengan nilai dari key.

1.6.2 Selection Sort

Impelementasikan algoritma selection sort dalam Java untuk mengurutkan serangkaian data integer. Lakukan percobaan terhadap hasil implementasi anda terhadap rangkaian data integer yang dimasukkan oleh pengguna melalui command line.

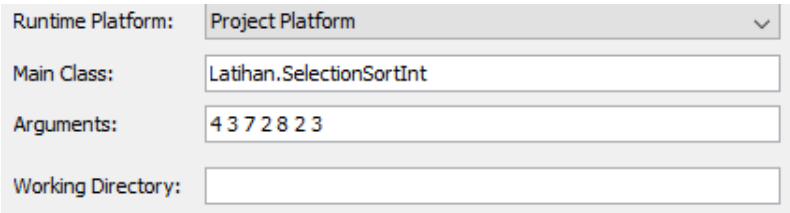
Program :

```
package Latihan;
public class SelectionSortInt {
    public static void selectionSort(int[] arr) {
        for (int i = 0; i < arr.length - 1; i++) {
            int index = i;
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[j] < arr[index]) {
                    index = j;
                }
            }
            int smallerNumber = arr[index];
            arr[index] = arr[i];
            arr[i] = smallerNumber;
        }
    }

    public static void main(String args[]) {
        int[] arr = new int[args.length];
        for (int i = 0; i < args.length; i++) {
            arr[i] = Integer.parseInt(args[i]);
        }
        System.out.println("Before Selection Sort");
        for (int i : arr) {
            System.out.print(i + " ");
        }
        System.out.println();

        selectionSort(arr);

        System.out.println("After Selection Sort");
        for (int i : arr) {
            System.out.print(i + " ");
        }
    }
}
```

Hasil :

```
Before Selection Sort
4 3 7 2 8 2 3
After Selection Sort
2 2 3 3 4 7 8 BUILD SUCCESSFUL (total time: 0 seconds)
```

Arguments :



Analisis :

Proses diatas diawali dengan user menginput data pada arguments melalui command-line. Kemudian program akan mencoba membuat array dengan panjang sebanyak data yang diinputkan. Kemudian data yang diinputkan tersebut akan dimasukkan kedalam array yang akan menyimpan data yang akan di sortir.

```
for (int i : arr) {
     System.out.print(i + " ");
}
```

- Proses diatas adalah forEach yang berguna untuk melakukan perulangan sebanyak nilai array dan akan mencetak data yang ada didalam array tersebut.

```
selectionSort(arr);
```

- kemudian program mencoba memanggil fungsi selectionSort yang diisi dengan data dari array yang ada.

```
for (int i = 0; i < arr.length - 1; i++) {
```

- pada fungsi insertionSort. Program mencoba melakukan perulangan selama nilai i = 0 dan i < dari panjang nilai array – 1.

```
int index = i;
```

- Membuat variabel index yang diisi dengan nilai dari i.

```
for (int j = i + 1; j < arr.length; j++) {
     if (arr[j] < arr[index]) {
         index = j;
     }
}
```

- Melakukan perulangan selama nilai j = i + 1, dan nilai j < dari panjang array.
- Kemudian melakukan pengecekan apakah nilai array pada index j lebih kecil dari nilai array pada index variabel index.
- Jika benar, maka akan menimpa variabel index dengan variabel j.

```
int smallerNumber = arr[index];
arr[index] = arr[i];
arr[i] = smallerNumber;
```

- Membuat variabel smallerNumber yang diisi dengan nilai dari array pada index variabel index.
- Mengisi data array pada index variabel index dengan data dari array pada index i.
- Mengisi data array pada index i dengan data dari variabel smallerNumber.

```
for (int i : arr) {
     System.out.print(i + " ");
}
```

- Kemudian mencetak seluruh hasil dari array yang telah disorting.

1.6.3 Merge Sort

Gunakan implementasi merge sort berikut ini terhadap serangkaian data integer.

```
public class MergeSortInt {
    static void mergeSort(int array[], int startIdx, int endIdx){
        if (startIdx == _____) {
            return;
        }
        int length = endIdx - startIdx + 1;
        int mid = _____;
        mergeSort(array, _____, mid);
        mergeSort(array, _____, endIdx);
        int working[] = new int[length];
        for (int i = 0; i < length; i++) {
            working[i] = array[startIdx + i];
        }
        int m1 = 0;
        int m2 = mid - startIdx + 1;
        for (int i = 0; i < length; i++) {
            if (m2 <= endIdx - startIdx) {
                if (m1 <= mid - startIdx) {
                    if (working[m1] > working[m2]) {
                        array[i + startIdx] = working[m2++];
                    } else {
                        array[i + startIdx] = _____;
                    }
                } else {
                    array[i + startIdx] = _____;
                }
            } else {
                array[_____] = working[m1++];
            }
        }
    }
```

```
    public static void main(String args[]) {
        int numArr[] = new int[args.length];
        for (int i = 0; i < args.length; i++) {
            numArr[i] = Integer.parseInt(args[i]);
        }
        mergeSort(numArr, 0, numArr.length - 1);
        for (int i = 0; i < numArr.length; i++) {
            System.out.println(numArr[i]);
        }
    }
}
```

Program :

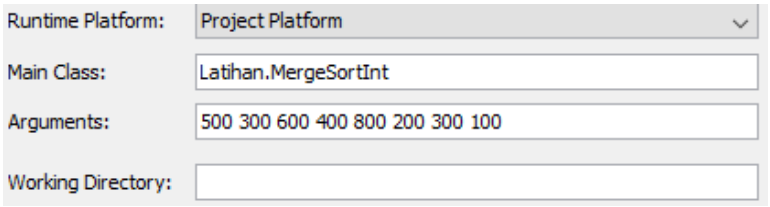
```
package Latihan;
public class MergeSortInt {
    static void mergeSort(int array[], int startIdx, int endIdx){
        if (startIdx == endIdx) {
            return;
        }

        int length = endIdx - startIdx + 1;
        int mid = (startIdx + endIdx) / 2;
        mergeSort(array, startIdx, mid);
        mergeSort(array, mid + 1, endIdx);
        int working[] = new int[length];

        for (int i = 0; i < length; i++) {
            working[i] = array[startIdx + i];
        }

        int m1 = 0;
        int m2 = mid - startIdx + 1;

        for (int i = 0; i < length; i++) {
            if (m2 <= endIdx - startIdx) {
                if (m1 <= mid - startIdx) {
                    if (working[m1] > working[m2]) {
                        array[i + startIdx] = working[m2++];
                    } else {
                        array[i + startIdx] = working[m1++];
                    }
                } else {
                    array[i + startIdx] = working[m2++];
                }
            } else {
                array[i + startIdx] = working[m1++];
            }
        }
    }
```

```
    public static void main(String args[]) {
        int numArr[] = new int[args.length];
        for (int i = 0; i < args.length; i++) {
            numArr[i] = Integer.parseInt(args[i]);
        }

        System.out.println("Before Sorting");
        for (int i : numArr) {
            System.out.print(i + " ");
        }

        mergeSort(numArr, 0, numArr.length - 1);
        System.out.println("\nAfter Sorting");
        for (int i : numArr) {
            System.out.print(i + " ");
        }
    }
}
```

Hasil :

```
Before Sorting
500 300 600 400 800 200 300 100
After Sorting
100 200 300 300 400 500 600 800 BUILD SUCCESSFUL (total time: 0 seconds)
```

Arguments :



Analisis :

```
int numArr[] = new int[args.length];
for (int i = 0; i < args.length; i++) {
     numArr[i] = Integer.parseInt(args[i]);
}
```

- Mengambil panjang dari args dan menjadikan sebagai panjang array.
- Melakukan perulangan untuk mengisi data yang ada didalam args kedalam array yang telah dibuat.

```
for (int i : numArr) {
     System.out.print(i + " ");
}
```

- Mencetak data array sebelum di sorting dengan forEach.

```
mergeSort(numArr, 0, numArr.length - 1);
```

- Memanggil fungsi mergeSort dengan mengisi argumen berupa data array, 0 sebagai menandakan index awal. Dan panjang data dari array – 1.

```
static void mergeSort(int array[], int startIdx, int endIdx){
```

- Membuat fungsi mergeSort dengan menerima beberapa parameter, yaitu parameter array, startIdx dan endIdx.

```
if (startIdx == endIdx) {
    return;
}
```

- Mengecek apakah nilai dari variabel startIdx sama dengan nilai endIdx. Jika sama maka akan langsung return.

```
int length = endIdx - startIdx + 1;
int mid = (startIdx + endIdx) / 2;
mergeSort(array, startIdx, mid);
mergeSort(array, mid + 1, endIdx);
int working[] = new int[length];
```

- Membuat variabel length yang diisi dengan nilai hasil dari endIdx – startIdx + 1.
- Membuat variabel mid yang diisi hasil dari (startIdx + endIdx) / 2.
- Memanggil fungsi mergerSort dengan mengisi argumen dari data array, startIdx, dan mid. Fungsi ini berfungsi sebagai rekursif.
- Memanggil fungsi mergeSort dengan mengisi nilai dari data array, mid + 1, dan endIdx. Fungsi ini juga berfungsi sebagai rekursif.
- Membuat variabel array working dengan panjang diinisialisasi dari nilai length.

```
for (int i = 0; i < length; i++) {
     working[i] = array[startIdx + i];
}
```

- Melakukan perulangan selama nilai i = 0 dan i < length.
- Maka akan mengisi data array working pada index i dengan data dari array pada index dari variabel startIdx + 1.

```
int m1 = 0;
int m2 = mid - startIdx + 1;
```

- Membuat variabel m1 yang menyimpan nilai 0.
- Membuat variabel m2 yang menyimpan nilai hasil mid – startIdx + 1.

```
for (int i = 0; i < length; i++) {
     if (m2 <= endIdx - startIdx) {
         if (m1 <= mid - startIdx) {
             if (working[m1] > working[m2]) {
                   array[i + startIdx] = working[m2++];
              } else {
                   array[i + startIdx] = working[m1++];
              }
         } else {
              array[i + startIdx] = working[m2++];
         }
    } else {
         array[i + startIdx] = working[m1++];
    }
}
```

- Melakukan perulangan selama nilai i = 0. Dan i < length.
- Mengecek apakah nilai m2 <= endIdx – startIdx. Jika true maka lanjut.
- Mengecek apakah nilai m1 <= mid – startIdx. Jika true maka lanjut.
- Mengecek apakah nilai array pada working dengan index dari nilai m1 lebih besar dari nilai array pada working dengan index dari nilai m2. Jika true maka lanjut.
- Mengisi nilai pada array di index i + startIdx dengan nilai dari working di index m2 + 1.
- Jika false, maka mengisi nilai pada array di index i + startIdx dengan nilai dari working di index m1 + 1.
- Jika `m1 <= mid – startIdx` menghasilkan false. Maka nilai pada array di index i + startIdx diisi dengan nilai dari working pada index m2 + 1.
- Jika `m2 <= endIdx – startIdx` menghasilkan false. Maka nilai pada array di index i + startIdx diisi dengan nilai dari working pada index m1 + 1.

```
System.out.println("\nAfter Sorting");
for (int i : numArr) {
     System.out.print(i + " ");
}
```

- Mencetak hasil data array setelah sorting menggunakan forEach.

1.6.4 QuickSort

Gunakan implementasi quicksort berikut ini terhadap serangkaian data integer.

```
public class QuickSortInt {
    static void quickSort(int[] array, int startIdx, int endIdx){
        // startIdx adalah index bawah
        // endIdx is index atas
```

```
        // dari array yang akan diurutkan
        int i = startIdx, j = endIdx, h;
        //pilih elemen pertama sebagai pivot
        int pivot = array[_____];
        // memilah
        do {
            while (array[i]_____pivot) {
                i++;
            }
            while (array[j] > _____) {
                j--;
            }
            if (i <= j) {
                h = _____;
                array[i] = _____;
                array[j] = _____;
                i++;
                j--;
            }
        } while (i <= j);
        // rekursi
        if (startIdx < j) {
            quickSort(array, _____, j);
        }
        if (i < endIdx) {
            quickSort(array, _____, endIdx);
        }
    }

    public static void main(String args[]) {
        int numArr[] = new int[args.length];
        for (int i = 0; i < args.length; i++) {
            numArr[i] = Integer.parseInt(args[i]);
        }
        quickSort(numArr, 0, numArr.length - 1);
        for (int i = 0; i < numArr.length; i++) {
            System.out.println(numArr[i]);
        }
    }
}
```

Program :

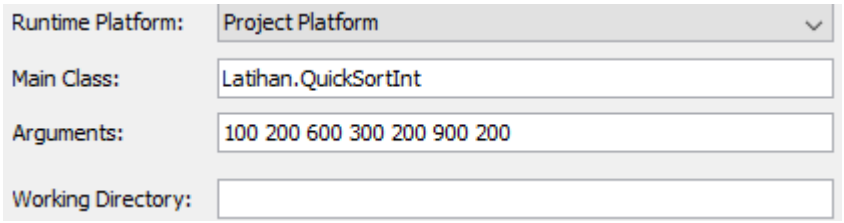
```
package Latihan;
public class QuickSortInt {
    static void quickSort(int[] array, int startIdx, int endIdx){
        // startIdx adalah index bawah
        // endIdx is index atas
        // dari array yang akan diurutkan
        int i = startIdx, j = endIdx, h;
```

```
        //pilih elemen pertama sebagai pivot
        int pivot = array[i];

        // memilah
        do {
            while (array[i] < pivot) {
                i++;
            }
            while (array[j] > array[startIdx]) {
                j--;
            }
            if (i <= j) {
                h = array[i];
                array[i] = array[j];
                array[j] = h;
                i++;
                j--;
            }
        } while (i <= j);
        // rekursi
        if (startIdx < j) {
            quickSort(array, startIdx, j);
        }
        if (i < endIdx) {
            quickSort(array, i, endIdx);
        }
    }

    public static void main(String args[]) {
        int numArr[] = new int[args.length];
        for (int i = 0; i < args.length; i++) {
            numArr[i] = Integer.parseInt(args[i]);
        }
        quickSort(numArr, 0, numArr.length - 1);
        for (int i = 0; i < numArr.length; i++) {
            System.out.print(numArr[i] + " ");
        }
    }
}
```

Hasil :

```
100 200 200 200 300 600 900 BUILD SUCCESSFUL (total time: 0 seconds)
```

Arguments :



Analisis :

```
int numArr[] = new int[args.length];
for (int i = 0; i < args.length; i++) {
     numArr[i] = Integer.parseInt(args[i]);
}
```

- Mengambil panjang dari args dan menjadikan sebagai panjang array.
- Melakukan perulangan untuk mengisi data yang ada didalam args kedalam array yang telah dibuat.

```
quickSort(numArr, 0, numArr.length - 1);
```

- Memanggil fungsi quickSort dengan mengisi argumen berupa data dari numArr, 0, dan panjang data array numArr – 1.

```
static void quickSort(int[] array, int startIdx, int endIdx){
```

- Membuat fungsi quickSort yang menerima beberapa parameter berupa array, startIdx dan endIdx.

```
int i = startIdx, j = endIdx, h;
int pivot = array[i];
```

- Membuat variabel i yang diisi dari nilai startIdx.
- Variabel j yang diisi dengan nilai endIdx.
- Dan variabel h dengan tipe data int.
- Membuat variabel pivot yang diisi dengan nilai array pada index i.

```
do {
    while (array[i] < pivot) {
          i++;
    }
```

- Melakukan perulangan do. Dan melakukan perulangan while yang mengecek apakah nilai array di index i lebih kecil dari nilai pivot.
- Jika true maka akan membuat i + 1.

```
while (array[j] > array[startIdx]) {
       j--;
}
```

- Melakukan perulangan selama nilai array pada index j lebih besar dari nilai array pada index startIdx.
- Jika true, maka akan membuat j – 1.

```
if (i <= j) {
    h = array[i];
    array[i] = array[j];
    array[j] = h;
    i++;
    j--;
}
```

- Melakukan pengecekan apakah i lebih kecil sama dengan j.
- Jika true, maka membuat nilai h disi dengan nilai dari array index i.
- Dan nilai array index i diisi dengan nilai array index j.
- Dan increment variabel i dan decrement variabel j.

```
} while (i <= j);
```

- Perulangan do diatas akan diulang selama nilai i lebih kecil sama dengan j.

```
if (startIdx < j) {
    quickSort(array, startIdx, j);
}
```

- Melakukan pengecekan apakah nilai startIdx lebih kecil dari nilai j.
- Jika true, maka akan memanggil fungsi quickSort dengan mengisi argumen berupa nilai array, startIdx, dan nilai j.
- Fungsi diatas adalah untuk melakukan rekursif.

```
if (i < endIdx) {
    quickSort(array, i, endIdx);
}
```

- Melakukan pengecekan apakah nilai i lebih kecil dari nilai endIdx.
- Jika true, maka akan memanggil fungsi quickSort dengan mengisi argumen berupa nilai array, i, dan endIdx.

```
for (int i = 0; i < numArr.length; i++) {
     System.out.print(numArr[i] + " ");
}
```

- Mencetak seluruh data dari array yang telah disorting sebelumnya ke layar.

KESIMPULAN

Insertion Sort merupakan metode pengurutan dimana langkah-langkahnya ialah membandingkan dari data kedua ke data pertama (Data-N <==> Data-N-1). Algoritma insertion sort pada dasarnya memilah data yang akan diurutkan menjadi dua bagian, yang belum diurutkan (meja pertama) dan yang sudah diurutkan (meja kedua).

Selection sort adalah algoritma untuk sorting sama seperti halnya dengan Insertion sort yang simpel, namun tidak efisien. Pada perulangan pertama, metode ini akan memilih elemen terkecil dari array, kemudian menukarnya dengan elemen pertama. Selanjutnya pada perulangan kedua, akan memilih nilai terkecil kedua (atau elemen dengan nilai terkecil diantara seluruh elemen yang tersisa), dan kemudian menukarnya dengan elemen yang kedua.

Algoritma pengurutan data merge sort dilakukan dengan menggunakan cara divide and conquer yaitu dengan memecah kemudian menyelesaikan setiap bagian kemudian menggabungkannya kembali. Pertama data dipecah menjadi 2 bagian dimana bagian pertama merupakan setengah (jika data genap) atau setengah minus satu (jika data ganjil) dari seluruh data, kemudian dilakukan pemecahan kembali untuk masing-masing blok sampai hanya terdiri dari satu data tiap blok.

QuickSort merupakan Algoritma pengurutan yang dikembangkan oleh Tony Hoare. performa rata-rata pengurutan O(n log n) untuk mengurutkan n item. Algoritma ini juga dikenal sebagai Partition-Exchange Sort atau disebut sebagai Sorting Pergantian Pembagi. Pada kasus terburuknya, algoritma ini membuat perbandingan $O(n^2)$, malaupun kejadian seperti ini sangat langka. Quicksort sering lebih cepat dalam praktiknya daripada algoritma merge sort dan heapshort. Dan juga, urutan dan referensi lokalisasi memori quicksort bekerja lebih baik dengan menggunakan cache CPU, jadi keseluruhan sorting dapat dilakukan hanya dengan ruang tambahan O(log n).

DAFTAR PUSTAKA

1. Bahasa Java. Algoritma Pemrograman Java Sorting : Insertion Sort. https://bahasajava.com/algoritma-pemrograman-java-sorting-insertion-sort/.
2. Yudi Setiawan. Metode Insertion Sort di Java Console. http://jagocoding.com/tutorial/812/Metode_Insertion_Sort_di_Java_Console/.
3. JavaTPoint. Insertion Sort in Java. https://www.javatpoint.com/insertion-sort-in-java.
4. Fidelson Tanzil., S.KOM., M.T.I. Insertion Sort. https://socs.binus.ac.id/2019/12/30/insertion-sort/. School of Computer Science Binus University.
5. Bahasa Java. Algoritma Sorting : Selection Sort. https://bahasajava.com/algoritma-sorting-selection-sort/.
6. Arfian Hidayat. Algoritma Selection Sort. 21 Mei 2019. https://arfianhidayat.com/algoritma-selection-sort.
7. Fidelson Tanzil., S.KOM., M.T.I. Selection Sort. https://socs.binus.ac.id/2019/12/26/selection-sort/. School of Computer Science Binus University.
8. JavaTPoint. Selection Sort in Java. https://www.javatpoint.com/selection-sort-in-java.
9. Creator Media. Pengertian Algoritma Divide and Conquer. https://creatormedia.my.id/rangkuman-algoritma-divide-and-conquer/.
10. Alfian Hidayat. Algoritma Merge Sort. 23 Mei 2019. https://arfianhidayat.com/algoritma-merge-sort.
11. JavaTPoint. Merge Sort. https://www.javatpoint.com/merge-sort.
12. Alza. Algoritma Quick Sort. 25 Maret 2019. https://koding.alza.web.id/algoritma-quick-sort/.
13. Dr. Suharjito, S.Si., MT. Simulasi Algoritma QuickSort. 19 Februari 2020. https://onlinelearning.binus.ac.id/computer-science/post/simulasi-algoritma-quicksort.
14. Sixv.com. Algoritma Quick Sort. https://www.sixv.com/algoritma/algoritma-quick-sort/.
15. AnakTik. Algoritma Quick Sort. https://anaktik.com/quick-sort/.
16. JavaTPoint. Quick Sort. https://www.javatpoint.com/quick-sort.